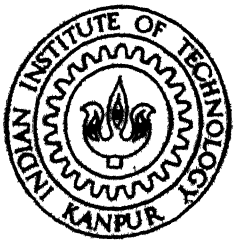
Enhancement Of GAtest

by

CHAPRAM. SUDHAKAR



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY. KANPUR

July, 1996

Enhancement Of GAtest

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Chapram.Sudhakar

to the

DEPARTMENT OF

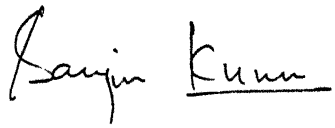
COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

July, 1996.

CERTIFICATE

This is to certify that the work contained in the thesis entitled "*Enhancement Of GAtest*" by "*Chapram.Sudhakar*" (Roll No: 9411108), has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.



Dr. Sanjeev Kumar Aggarwal
Associate Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

Dr. R.K. Ghosh
Associate Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

July, 1996

Acknowledgments

I am grateful to my thesis supervisors Dr. S.K. Aggarwal and Dr. R.K. Ghosh for their invaluable guidance and encouragement through out this work. I am thankful to Devireddy Dilip Kumar and Meda Lakshmi Prasanna Kumar Reddy for their advice through out my work. I am thankful to Jaya Ram, Srivatsa for their concern towards me. I would like to thank all of my friends in M.Tech. 94 and M.Tech. 95 for making my stay at I.I.T. Kanpur, a memorable one. Finally I would like thank my parents and sisters and Anil and Narmada for their love and encouragement.

Abstract

Dependence analysis is an important part of a restructuring compiler. Many tests are proposed for dependence analysis. Most of them are not accurate. The accurate tests are more complicated. A simple but exact dependence test based on genetic algorithms was proposed earlier. But it was applicable to only problems with known bounds. This test is adapted to the dependence problem with unknown bounds. **WHILE** loops and certain **DO** loops with exit statements are generally treated sequential programs. The problem in parallelizing **WHILE** loops is, that iteration space is unknown. But some **WHILE** loops are parallelizable. Currently there is no test that can analyze dependence problem of **WHILE** loops at compile time. GAtest is extended to solve some cases of dependence problem of **WHILE** loops.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The major objective in research and development of parallel processing system is to extract high performance for application that may require big number crunching capabilities or manipulation of huge data. There are two simple but key ideas which lead us to realize parallelism in computer architectures:

1. **Separating functionality:**

   The processing of a single instruction is partitioned into several parts or functional units which are performed by different structural units of ALU. So at any time distinct parts of a number of instructions can be executed in parallel. The parallelism of this kind is called *pipeline* parallelism.

2. **Duplicating parts and separate application:**

   Parts of a computer system are replicated. For example, if ALUs are duplicated we have *super scalar* architecture. Replication of CPUs give *multiprocessor* system. Similarly whole computer system can be replicated and all the systems interconnected by a network. Such an environment, can be homogeneous or heterogeneous, provides a *PVM* (Parallel Virtual Machine) like computing environment. Also application's parts are separated to run on these replicated parts in parallel.

It, clearly, indicates that several factors have to be considered in designing high performance computer system. The algorithms should be carefully designed to

match architectures and programming environments.

The other important factor which impedes effective use of parallel computing system is *software inertia*. The investment in developing software for sequential computers is quite enormous. The manual re-engineering of these software to run on parallel system is not only infeasible but too expensive. Therefore what is needed is a program *re-structurer*, which will take sequential programs as input and produce optimized equivalent flow correct parallel program, without or with least human intervention. In fact, parallel computers have remained in research and development stage for so long due to lack of efficient code restructuring technology.

1.1 Restructuring compiler

A restructuring compiler takes a sequential program as input and produces an equivalent parallel program for a target parallel machine as output by performing number of transformations to extract the parallelism. Most of the restructuring compilers have been developed as research tools and work on source code level. Besides standard compiler optimization techniques, viz., *loop fusion*, *strip mining*, *loop interchange*, new transformations have been developed specifically for restructuring sequential code for execution on a target parallel machines. A restructuring compiler has to analyze the memory references to determine which transformations may be applied. The transformations must preserve the flow correctness of the original program. This can be achieved by ensuring that memory references are consistent by imposing dependence relationships, among references. The flow correct parallel programs will also be logically correct if the corresponding sequential program itself is logically correct.

The execution order of statements in a sequential program preserves the dependence relations. However when a number of these statements have to be executed in parallel, the dependence relations among the memory references by the statements may be violated. In order to understand to what extent the dependence relations are violated and what techniques should be applied to resolve or eliminate the dependencies it is essential to understand the concept of dependence relations.

1.2 Data dependence

Let **S** and **T** be pair of statements selected arbitrarily from a sequential program. The set of variables written by a statement is called *define* set of that statement and similarly set of variables read is called *use* set of that statement. Statement **T** is dependent on **S** if there exist some memory location **M**, such that

1. **S** and **T** both reference **M** and **M** belongs to define set of **S** or **T** or both,
2. **S** precedes **T** in sequential execution of program,
3. **M** is not written in between **S** finishes and **T** starts.

Three types of dependencies exist based upon the references to **M** by **S** and **T**, namely

- **T** is *flow-dependent* on **S** if **S** writes **M** and then **T** reads it,
- **T** is *anti-dependent* on **S** if **S** reads **M** and then **T** writes it,
- **T** is *output-dependent* on **S** if **S** writes **M** and then **T** writes it again.

We use **S** δ_f **T** to denote flow dependence, **S** δ_a **T** for anti-dependence and **S** δ_o **T** for output dependence.

Example 1.1:

S1: a = b + c	{S1, a} δ_o {S2, a} - Output Dependency
S2: a = b + d	{S2, a} δ_f {S3, a} - Flow Dependency
S3: c = sqrt(b) + a	{S1, c} δ_a {S3, c} - Anti Dependency

∎

The above dependencies are explained at the granularity of statements for simplicity. In many applications, mainly in scientific problems, the loops constitutes a significant portion of code. An empirical estimate predicts the program control is localized in various loops for about 80% of the time. Therefore, to exploit the fine grain parallelism in a program the main target should be for loop parallelization. Loop parallelization means to restructure the code inside the loops in such a way

that several iterations of loop can be executed in parallel. However, loop restructuring is possible if there are no data dependences between iterations. For example, consider the following piece of code.

```
a(0) = c
for (i=1; i<N; i++)
   a(i) = a(i-1) + 2
```

In first iteration value of `a(1)` is determined using the value of `a(0)`. Similarly in second iteration value of `a(2)` is determined using `a(1)` and so on. This means unless the $(i-1)$th iteration of loop is complete ith iteration cannot start. The dependences like this, where the value generated by a previous iterations of loop is required for generating a value in a subsequent iteration is called *loop carried dependences*. All loop carried dependences have to be either eliminated or preserved before restructuring a loop for parallel execution.

The problems associated with data dependences for array variables are fundamental to the results of investigations presented here. These problems have, therefore, been discussed in more details in the next chapter. There are many dependence tests [13, 2, 16, 8, 14] to solve the dependence problem. A general solution to dependence problem requires integer programming which takes exponential time. But several tests suited for some particular class of problems were found. Most of these tests are simple and conservative. Omega test [14] is the exact test found so far. But in worst case it may take exponential time. None of the existing tests are suited for analyzing loops with non-constant increments. Similarly, barring omega test, none of the tests is applicable to solution of non-linear equation. In this thesis a special case of dependence problem involving non-constant loop increment, recurrence relation, is solved using genetic algorithms. It is used for **WHILE loop** parallelization. The technique can be extended to the case of general **for** loops of C programs in a straight forward way. Though the solution takes exponential time in worst case, in practice it is found to be quite fast as indicated by the experimental results.

1.3 Overview of thesis

The work done in this thesis can be divided into three parts. The first part is a comprehensive study of the various dependence tests and their performance evaluations. Around 50 real programs written by different users including students and faculty were chosen. Length of these programs range from 20 to 1700 lines. The results of the performance study are presented in Chapter 2. The second part is essentially enhancements to GAtest [12] which is presented in third chapter. The enhancement consists of the following.

- Testing for a particular direction vector.
- Applicability of GAtest to unknown bounds.
- A new concept of S^2-*spaces* (See Section 3.8) is introduced.

GAtest has been adapted for S^2-space. The third part is connected with the parallelization of **WHILE** loops. GAtest has been adapted to solve dependence problem in case of **WHILE** loops.

Chapter 2

Dependence Tests

A sequential program can be considered as a sequence of statements containing a single thread of control in execution. Most programs are deterministic in which any portion of thread contains a single entry and a single exit point. Loop is a programming construct which is associated with a portion of thread of control in runtime environment, in which control passes through the same path repeatedly for a finite number of times.

Example 2.1: Program 2.1.

```
S1: j = 0
    for i=1 , 10 do
S2:   a(i) = i
S3:   b(i) = i + 1
      endfor
S4:   print "end"
```

Program thread : S1 , $(S2 , S3)^{10}$, S4

Loop : $(S2 , S3)^{10}$

∎

In the above example the notation S^n is used to denote that S is repeated n times.

A parallel program can be considered as a set of statements with more than one thread of control in execution. The loops have major potential for preparing two or more threads of an equivalent parallel program from a corresponding single thread of the sequential program. This leads to parallelizing the loop. Parallel program should synchronize at the points, where dependence exist among threads. If there is

no dependence among threads, synchronization is needed only at the entry and exit points of the loop, resulting in parallel execution of different iterations of a loop.

Example 2.2: Program 2.2. The equivalent parallel program of example Program 2.1 is shown below

```
S1:  j = 0
     forall i=1 , 10 do
s2:     a(i) = i
s3:     b(i) = i+1
     endfor
s4:  print "end"
```

∎

The threads of execution of a loop can be represented graphically, called *iteration space diagram.* It is a N-dimensional plot, in which each dimension represents an index variable of a loop, each point represents an iteration in execution and an arrow between points indicates dependence between iterations. The dependence between two statements of different iterations of a loop is called loop carried dependence. A more detailed diagram can be drawn using N+1 dimensions, extra dimension indicating execution in a single iteration. In addition to loop carried dependences, this diagram also shows dependencies between statements of a single iteration called *loop invariant dependencies.*

Example 2.3: Program 2.3

```
     for i=1 , 4 do
S1:     a(i) = 2 + b(i)
S2:     b(i) = a(i-1) + a(i+1)
     endfor
```

∎

Loop invariant dependencies cause no problem to loop parallelization. But loop carried dependencies have to be eliminated or must be preserved by transformations like *loop splitting*, *introducing new variables*, etc. The dependence problem of loops is formally defined in the following section. In case of dependence in loops certain

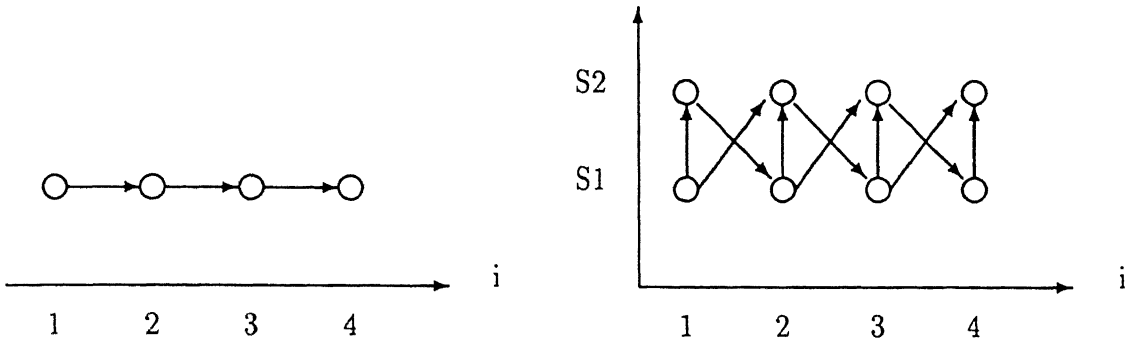


Figure 1: Iteration space diagram of Prog 2.3

issues such as *direction of the dependence* and *distance* between dependent iterations come into picture. As mentioned in the Chapter 1 several tests have been proposed for the dependency problem of loops. A comprehensive but brief study of some well known data dependence tests have been presented in this chapter.

2.1 Dependence problem of loops

Let **S** and **T** be a pair of statements in a nest of loops. **T** is dependent on **S** if there exist instances S' and T' of **S** and **T** respectively in the unrolled loop such that T' is dependent on S'.

Consider following two array references within a loop nest of depth n.

```
for I1 = 1 to N1 do
for I2 = 1 to N2 do
   ...
for In = 1 to Nn do
   X(f1(I1,I2,...,In),f2(I1,I2,...,In),...,fn(I1,I2,...,In)) = ...
   ... = X(g1(I1,I2,...,In),g2(I1,I2,...,In),...,gn(I1,I2,...,In))
endfor
   ...
endfor
endfor
```

If the above two references of X are referring to the same memory location for some iterations $\mathbf{I} = (i_1, i_2, \ldots, i_d)$ and $\mathbf{J} = (j_1, j_2, \ldots, j_d)$ then

$$\left.\begin{array}{rcl} \mathtt{f1}(\mathbf{I}) & = & \mathtt{g1}(\mathbf{J}) \\ \mathtt{f2}(\mathbf{I}) & = & \mathtt{g2}(\mathbf{J}) \\ & \vdots & \\ \mathtt{fn}(\mathbf{I}) & = & \mathtt{gn}(\mathbf{J}) \end{array}\right\} \quad (1)$$

and $1 \leq i_k, j_k \leq N_k, 1 \leq k \leq n$.

In order to determine **T** is dependent on **S**, it must be proved that at least a pair of **I**, **J** exists that satisfy the above equations. If all `fi` and `gi` are linear functions then each of the equations in the set of equations (1) is also a linear equation. In general a linear equation in n variables with the variables being bounded can be represented as follows.

$$a_1 I_1 + a_2 I_2 + \cdots + a_n I_n = a_0 \quad (2)$$

$$\left.\begin{array}{rcl} l_1 \leq & I_1 & \leq u_1 \\ l_2 \leq & I_2 & \leq u_2 \\ & \vdots & \\ l_n \leq & I_n & \leq u_n \end{array}\right\} \quad (3)$$

Equation (2) is called linear diophantine equation.

2.2 Direction vector

Let **S** and **T** be two statements inside a loop nest and **T** is dependent on **S**. This implies that there are two iterations $\mathbf{I} = \{i_1, i_2, \ldots, i_d\}$ and $\mathbf{J} = \{j_1, j_2, \ldots, j_d\}$ for which statements **S** and **T** refer to a common memory location. The direction vector of this dependence relation can be defined as

$$\mathbf{D} = \{sig(j_1 - i_1), sig(j_2 - i_2), \ldots, sig(j_d - i_d)\}$$

where

$$sig(j-i) = \begin{cases} '=' & \text{if } i = j; \\ '<' & \text{if } i > j; \\ '>' & \text{if } i < j. \end{cases}$$

In the direction vector, if a component is $'='$ then it indicates a loop invariant dependence, with respect to the loop of that component. Similarly if a component is $'<'$ or $'>'$, then it indicates loop carried dependence, with respect to the loop of that component. A component can have multiple symbols, if different instances of the two statements are having different type of dependencies. If for a particular component all directions exist then it can be replaced by a $'*'$. i.e. If $(<,=)$, $(<,<)$ and $(<,>)$ direction vectors are valid, these can be denoted by one direction vector $(<,*)$.

2.3 Dependence distance

The distance between the dependent iterations of a loop in terms of number of iterations is the dependence distance.

If $\mathbf{I} = (i_1, i_2, \ldots, i_d)$ and $\mathbf{J} = (j_1, j_2, \ldots, j_d)$ are the iterations that are causing dependence then the distance vector is

$$\mathbf{DD} = \{j_1 - i_1, j_2 - i_2, \ldots, j_d - i_d\}.$$

The distance vector may vary from one instance to another instance of the dependence of the same two statements. Dependence distances are required for vectorization of serial programs and converting serial **do loops** to **doacross** loops.

2.4 Dependence tests

Many tests were proposed to solve the dependence problem. Unfortunately, none of these tests can handle all the cases that arise in a dependence problem. Some tests are best suited to a particular class of problems. Most of the tests are conservative in nature in the sense that they predict dependency even in the cases where it may not exist at all. Some of the important tests are studied in this section.

2.4.1 GCD test

GCD test is the simplest test of all the dependence tests. It is based on elementary theorem of number theory stating that the Equation (2) has an integer solution if and only if $d = gcd(a_1, a_2, \ldots, a_n)$ divides a_0. This is extended by combining with Gaussian elimination method, to be applicable to a set of linear diophantine equations [13].

The system of linear diophantine equations can be represented in the matrix form

$$XA = C.$$

GCD test finds a unimodular matrix U and echelon matrix D such that $UA = D$. Algorithm to find unimodular matrix is given below.

Algorithm 2.1: GGCD.

```
Generalised_GCD(CoefMatrix) {
   for (i=1; i<=NoColums; i++) {
      From row i+1 to all rows make all elements in column i
      zeros by applying elementary row transformations.

      Apply the same elementary row transformations to the
      unit matrix.
   }
   return (unimodular matrix = result of transforming unit matrix)
}
```

∎

If an integer matrix T exists satisfying $TD = C$ then $X = TU$ is a solution to the system. Since D is an echelon matrix the equation $TD = C$ can be solved easily. The drawback of this test is it doesn't consider the bounds at all. So if the gcd of the coefficients is 1 then this test always predicts dependence, even though there is no integer solution within the given range of indices.

Example 2.4: The following example is reproduced from [13]. Consider the following equations.

$$3i_1 - 2i_2 - 2j_2 = 2$$

$$i_1 - 2i_2 = 1$$

Coefficient matrix A is

$$\begin{pmatrix} 3 & 1 \\ -2 & -2 \\ -2 & 0 \end{pmatrix}$$

Applying above algorithm we get unimodular matrix

$$D = \begin{pmatrix} 1 & 1 \\ 0 & -2 \\ 0 & 0 \end{pmatrix}$$

The equation $TD = C$ becomes

$$\begin{pmatrix} t_1 & t_2 & t_3 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & -2 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 \end{pmatrix}$$

This gives $t_1 = 2$ and $t_1 - 2t_2 = 1$. There is no integer solution to t_2. Therefore the system of equations has no solution.

■

2.4.2 Banerjee's test

Banerjee's test [13] is based on intermediate value theorem stating that, the linear diophantine equation (2) has a real solution within the bounds (3), if $b_{low} \leq a_0 \leq b_{up}$, where b_{low} and b_{up} are lower and upper bounds of expression $\sum a_i I_i$ respectively, within the region bounded by (3). This test computes the bounds of the linear expression $\sum a_i I_i$ and if a_0 is within these bounds it predicts dependency. The bounds of a linear expression in a rectangular space can be calculated as follows. Lower bound b_{low} occurs when

$$I_i = \begin{cases} l_i & \text{if } a_i \geq 0 \\ u_i & \text{if } a_i < 0 \end{cases}$$

and upper bound b_{up} occurs when

$$I_i = \begin{cases} u_i & \text{if } a_i \geq 0 \\ l_i & \text{if } a_i < 0. \end{cases}$$

where l_i, u_i are bounds of variable I_i.

In case of trapezoidal regions bounds can be calculated from the algorithm given below.

Algorithm 2.2: Banerjee.

```
Banerjee(P , Q , Dioph)
P , Q : Lower and upper bounds
Dioph : Diophantine equation
{
   Initialize.  {
      blow = 0
      bup  = 0
      k    = n
      D    = Dioph
      E    = Dioph
   }
   while (k >= 1) {
      Eliminate Ik {
         if (D[k] >= 0) blow = blow + D[k] * P[k][0]
         else blow = blow + D[k] * Q[k][0]

         if (E[k] >= 0) bup = bup + E[k] * Q[k][0]
         else bup = bup + E[k] * P[k][0]

         if (k > 1) {
            for (i=1; i<k; i++)   {
               if (D[k] >= 0) D[i] = D[i] + D[k] * P[k][i]
               else D[i] = D[i] + D[k] * Q[k][i]
```

```
                if (E[k] >= 0) E[i] = E[i] + E[k] * Q[k][i]
                else E[i] = E[i] + E[k] * P[k][i]
            }
        }
    }
    return (blow , bup)
}
```

■

This test is conservative in nature. Even though it indicates real solutions within the bounds there may not exist integer solutions.

Example 2.5: Consider the bounds of $4x + 3y$ in the region

$$\begin{aligned} 1 \leq\ & x\ \leq 3 \\ x \leq\ & y\ \leq 5 - x \end{aligned}$$

Lower and upper bounds of $4x + 3y$ from the above algorithm are 4 and 18 respectively. But 18 occurs at point (3 , 2) which will not be in the region.

■

From the above example it is known that in case of trapezoidal region bounds calculated from the above algorithm need not be within the region.

2.4.3 I-test

I-test [15] extends both the applicability and accuracy of the GCD and Banerjee's tests. Like GCD test it checks for the presence of integer solutions and like Banerjee's test it also considers the bounds of the variables, even if bounds of sufficiently many variables are unknown. It often produces definitive positive than GCD and Banerjee's tests. I-test changes the diophantine equation into an interval equation.

$$a_1 I1 + a_2 I_2 + \cdots + a_n I_n = [L, U] = [a_l, a_u] \tag{4}$$

Initially $a_l = a_u = a_0$.

Interval GCD test: Let $a_1, a_2, \ldots, a_n$ be integers, and let $d = gcd(a_1, a_2, \ldots, a_n)$, the interval equation (4) has an integer solution if and only if $L \leq d\lceil L/d \rceil \leq U$.

If the given diophantine equation satisfies the interval GCD test, some i such that $|a_i| \leq a_u - a_l + 1, 1 \leq i \leq n$ is chosen and by eliminating that variable new interval is formed as follows.

$$\begin{aligned} a'_l &= a_l - max(a_i x_i) \\ a'_u &= a_u - min(a_i x_i) \end{aligned}$$

This elimination of variables is repeated until further elimination of variables is not possible or interval GCD test fails.

Algorithm 2.3: I-test

```
Itest(Dioph , Constraints) {
   L = a0
   U = a0
   coeff = {a1 , a2 , ... , an}
   unknown = {all ai's whose bounds are unknown}

   if (unknown != NULL) {
      u = gcd(unknown)
      if (u == 1) return(TRUE)
   }
   coeff = (coeff - unknown) + u
   while (TRUE) {
      while (There is ai whose limits are known
            and abs(ai)<=U-L+1) {
         Select ai such that its limits are known
            and abs(ai)<=U-L+1
         L = L - max(ai*xi)
         U = U - min(ai*xi)
         coeff = coeff - ai
         if (coeff == NULL) {
            if (L <= 0 and 0 <= U)
               return(TRUE)
```

```
            else return(FALSE)
         }
      }
      d = gcd(coeff)
      if (!(L <= d ceil(L/d) <= U)) return(FALSE)
      if (d != 1) {
         for all ai in coeff ai = ai / d
         L = ceil(L/d)
         U = floor(U/d)
         if (L > U) return(FALSE)
      }
      else return(maybe)
   }
}
```

■

Example 2.6: This is reproduced from [15]. Consider the equation

$$I_1 - 3I_2 + 7I_3 = 8$$

subject to the limits

$$1 \leq I_1 \leq 3,\ 1 \leq I_2 \leq 2,\ 1 \leq I_3 \leq 4.$$

From GCD test gcd(1 , -3 , 7) = 1 divides 8. From Banerjee's test limits are (2 , 28). 8 is within the limits so both GCD and Banerjee's tests assumes dependence.

If we rewrite the equation as interval equation,

$$I_1 - 3I_2 + 7I_3 = [8, 8]$$

and eliminate I_1 we get

$$-3I_2 + 7I_3 = [8 - 3, 8 - 1] = [5, 7].$$

Interval GCD test indicates there may be a solution. Now we can eliminate I_2, which gives

$$7I_3 = [5 + 3, 7 + 6] = [8, 13]$$

Final application of interval GCD test reveals that there is no solution.
∎

2.4.4 Delta test

Delta test [2] is a multiple subscript exact test to be used with common coupled subscripts. All the subscripts of the two array references are partitioned into *minimal coupled groups*. For each group of subscripts this test is applied. If any group does not have a solution then the whole system has no solution. The main idea behind the delta test is, constraints derived from the SIV subscripts will be propagated into other subscripts in the same coupled group. Delta test says independence if any of the component ZIV or SIV tests determine independence. Otherwise it converts all SIV subscripts into constraints and propagates them into the MIV subscripts. This is repeated until no new constraints are found. Then constraints propagated for coupled RDIV (Restricted Double Index Variable) subscripts [2]. Finally remaining MIV subscripts are tested. The results are intersected with existing constraints. If the result of intersection is the empty set then no dependence exists.

Algorithm 2.4: Delta

```
DeltaTest(Subscripts) {
   Initialize elements of constraint vector CV = NULL
   while (untested SIV subscripts exists) {
      apply SIV test to all untested SIV subscripts
      return (FALSE) or
      derive new constraint vector NCV
      NCV = Intersection(NCV , CV)
      if (NCV == NULL) return (FALSE)
      else if (CV != NCV) {
         CV = NCV
         propagate constraint CV into MIV subscripts
            possibly creating new SIV or ZIV subscripts
         apply ZIV test to untested ZIV subscripts
         return (FALSE) or continue
```

```
        }
    }
    while (untested RDIV subscripts exists)
        test and propagate RDIV constraints

    test remaining MIV subscripts
    intersect resulting direction vectors with CV
    return (FALSE) or direction vector
}
```

■

Example 2.7: Consider the following example.

```
        DO 10 i
            DO 10 j
10              A(i+1 , i+j) = A(i , i+j)
            ENDDO
        ENDDO
```

■

Applying the strong SIV test to first subscript of array A derives the constraint $(i + 1 = i')$ and distance of 1. Propagating this constraint into second subscript $(i + j = i' + j')$ to eliminate i' gives $(j, j' + 1)$. Once again applying SIV test to the resulting subscripts gives distance vector of -1. Merging these to distances gives distance vector: (1 , -1).

2.4.5 Lambda test

The λ test is an approximate test which extends Banerjee's test for multidimensional array references. Consider the set of equations (1) of Section 2.1. If the equations are solved independently all may have integer solutions. However, simultaneously when the set is treated as a set of equations there may not be any integer solutions. That is the intersection of solution sets may be null. λ test solves the problem by

formulating it as a less exact problem,

$$\lambda_1(f_1(I) - g_1(J)) + \cdots + \lambda_n(f_n(I) - g_n(J)) = 0.$$

If a integer tuple $(\lambda_1, \lambda_2, \ldots, \lambda_n)$ not satisfying the above equation is found , then there is no integer solution to the system of linear equations.

Example 2.8: Consider the loop nest given below

```
DO I = 1 , 50
   DO J = 2 , 50
      A(I , I+1) = A(3*I+j+1 , I+j)
   ENDDO
ENDDO
```

To test for the flow dependence, linear equations to be solved are

$$x_1 - 3x_3 - x_4 = 1$$

$$x_1 - x_3 - x_4 = -1$$

with the constraints $1 \leq x_1, x_3 \leq 50$ and $2 \leq x_4 \leq 50$. If (-1 , 1) is chosen as the λ tuple we have

$$-(x_1 - 3x_3 - x_4) + (x_1 - x_3 - x_4) = -2$$

The above equation implies $x_3 = -2$ which is violating the constraint $1 \leq x_3 \leq 50$. So there is no dependence.

∎

2.4.6 Power test

Power test [8] works in two phases. It uses the *extended GCD* algorithm in the first phase. In addition to whether there are any integer solutions for the dependence equations without considering the loop limits, the extended GCD algorithm indicates formulas that can be used to specify the index variables in terms of free variables derived from the matrix equation

$$X = TU,$$

where U is the unimodular matrix resulted from the GCD test. Power test constructs lower and upper bounds for each of the free variables of the above matrix equation using loop bounds. These give boundaries to the solution space. *Fourier-Motzkin* variable elimination method, modified to find integer solutions, is used to test for an empty solution space. If the solution space is not empty then there exists integer solution.

Algorithm 2.5: Power

```
PowerTest(CoefMatrix , Constraints) {
   Apply Generalized GCD test
   Solve the Matrix equation X = TU to
      give index variables in terms of free variables.

   Substitute these index variables into
      constraints which gives constraints of free variables.
   Validate the resulting solution space using
      Fourier-Motzkin variable elimination method.
   return (FALSE) or direction vector
}
```

■

Fourier-Motzkin variable elimination method is given in Subsection 2.4.7.

Example 2.9: This example is reproduced from [8]. Consider the following program segment.

```
for I = 1 to N do
   for J = I + 1 to N do
      A[i] = A[j]
   endfor
endfor
```

The dependence equation is $i_1 - j_2 = 0$, the dependence matrix equation $XA = C$ is,

$$(i_1\ j_1\ i_2\ j_2)\begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix} = (0)$$

Extended GCD algorithm gives,

$$\begin{aligned} i_1 &= t_1 \\ j_1 &= t_3 \\ i_2 &= t_4 \\ j_2 &= t_2 \end{aligned}$$

From the loop limits we derive the following bounds on the free variables:

$$\begin{aligned} 1 \leq\ & t_2 \leq N \\ 1 \leq\ & t_3 \leq t_2 - 1, N \\ t_2 + 1 \leq\ & t_4 \leq N \end{aligned}$$

Since these are consistent there exists integer solution. Now suppose we want to test for particular dependence direction, such as (<) direction in the first loop. From $i_1 < j_1$ we derive the additional bound

$$t_2 + 1 \leq t_3.$$

This is inconsistent with the previous bounds, so there is no dependence with (<) direction in the first loop.

∎

2.4.7 Omega test

Omega test is an exact dependence test. Input to the omega test is a set of linear equalities ($\sum a_i x_i = c$) and set of linear inequalities ($\sum a_i x_i \geq c$). Omega test first eliminates all equality constraints producing a new problem of inequality constraints

that has integer solution if and only if the original problem had integer solutions. Then it checks for contradictory pair of inequalities. If such pair is found then no solution exists. Tight inequalities are converted to equalities and are eliminated as in the original problem. Redundant constraints are eliminated by tightening them. If the problem involves at most one variable and passed above tests then integer solution is reported. Otherwise Fourier-Motzkin variable elimination method is used to validate the inequalities. This method eliminates a variable from the linear programming problem. This projects a n dimensional object into $n-1$ dimensional shadow. The shadow that came through the object of at least depth 1 is called a dark shadow. Algorithm to find shadows and validating the inequalities is given below. Curious reader can refer [14] for more details.

Algorithm 2.6: Omega

```
Omega(equalities , inequalities) {
   do {
      Normalize the problem. If error return fail to callee.
      Determine  unit variable  or  least co-efficient variable.
      Eliminate unbounded variables'  inequalities, recursively.
      Eliminate redundant constraints and add a new constraint.
      If inconsistencies found, return fail to callee.
   }
   while (there are equalities)
   Fourier_Motzkin(resulting_inequalities)
}

Fourier_Motzkin(inequalities) {
   Select a least coefficient variable.
   Calculate real shadow
   Calculate dark shadow
   If (projection is exact)
      call omega on this shadow.
   else {
```

```
      If (there are no integer solutions to the real shadow)
         return (FALSE)
      If (there are integer solutions to the dark shadow)
         return (TRUE)
      Apply integer programming.
   }
}

Find_Shadows() {
   z = least_coefficient_variable of problem
   Classify all inequalities into two classes {
      I.  Lower bounds on z. /* of type e1 <=a*z
      II. Upper bounds on z. /* of type a*z >=e2
   }

   Calculate light shadow as follows {
      copy main problem into subproblem P1,
      Eliminate all inequalities, containing z.
      for (each (i , j)) {
            /* i in class I and j in class II */
            /* ith inequality: e1<=a*z        */
            /* jth inequality: e2>=b*z        */
         new_inequality: b*e1 <=a*e2;
         Insert  new_inequality into P1.
      }
   }
   Calculate dark  shadow as follows {
      copy main problem into subproblem P2,
      Eliminate all inequalities, containing z.
      for (each (i , j)) {
            /* i in class I and j in class II */
```

```
            /* ith inequality: e1<=a*z        */
            /* jth inequality: e2>=b*z        */
          new_inequality: a*e2-b*e1>=(a-1)(b-1)
          Insert  new_inequality into P2.
        }
      }
    }
```

2.5 Performance results

Tiny tool developed by Wolfe [9] is used to study the performance of various tests. Around 50 programs written by different users are studied. These programs are converted to tiny format, to make them acceptable by the tiny tool. This converter, *f2t-converter*, is also an integrated part of the tiny tool. Converted fortran programs are not acceptable to tiny because of **I/O** statements and **GOTO** statements and some others. So these programs are again filtered to produce programs containing only **do**, **if** and assignment statements.

Suresh [11] has also done the analysis of various tests using test suites of programs containing a total of around 200 loops. The following tables are summarized results of present study and the work presented by Suresh. The results of the study are tabulated below. Test suites named NASA NSA and Convex are used by Suresh and My-test suite is used for the present performance study of the tests.

From the Table 2 it is observed that many of the array references(73.35%) are linear. The non-linear references are mostly resulted from one single program (count 6201) in case of My-test suite. Surprisingly most of the coefficients of the variables in subscript expressions are zeros (74.89%) and none of the references is containing coefficients other than 0 and ± 1 in My-test suite. In other suites also number of non-unit and non-zero coeffiecients is negligible. Many programs are using one dimensional arrays(92.96%). No program is found having more than 2-dimensional arrays in both Convex and My-test suites.

Comparative performance of the various tests is given in Table 3. Among all the tests only Omega test is performed well. All other tests are failed due to the symbolic constants passed to subroutines. Banerjee's test by taking much less time performed well compared to other tests, excluding Omega test. It is understandable from the presentation of Power test, even though generally it performs well, here it failed due to the unknown variables in the bounds. Power test is able to identify more independent than generalized GCD test alone. Time required for the Omega test is around 18 times that of Banerjee's test.

From the results of the experiments it can be concluded that a restructuring compiler needs a test that efficiently tests the dependence problem involving linear coefficients and particularly unknown bounds.

Table 1: Coefficients of subscripts expressions

Test suite	*0*	± 1	*Others*
My-test	123391	45085	0
NASA NSA	49784	12468	12
Convex	1123	873	10
Percentage	74.89%	25.10%	0.01%

Table 2: Types of subscripts

Test suite	*Linear*	*Non-Linear*
My-test	17113	7051
NASA NSA	3130	430
Convex	409	24
Percentage	73.35%	26.65%

Table 3: Frequency of array dimensions

Test suite	*Dim-1*	*Dim-2*	*Dim-3*	*Dim-4*
My-test	15561	8603	0	0
NASA NSA	355	792	1951	462
Convex	400	34	0	0
Percentage	92.96%	5.61%	1.16%	0.27%

Table 4: Performance analysis of dependence tests

Test	*Applied*	*Independence*	*Time in ms*
Banerjee	1437	31	5940
GGcd	1437	19	5240
λ	1437	21	94360
Power	1437	37	10960
Delta	1437	35	32340
Omega	1409	1365	107760

Chapter 3

GA Test

Holland proposed genetic algorithms in early 1970s as guided randomized search procedures that mimic the evolutionary process in nature. Genetic algorithms manipulate a population of potential solutions to a problem. They operate on encoded representations of the solutions equivalent to the genetic material of the individuals in the nature. Each solution is associated with a fitness value that indicates its goodness of fit. The various genetic operators e.g. crossover cause the information exchange among the members of the population. These genetic operators coupled with the survival of the fittest strategy are employed to locate better solutions and ultimately to solve the problem.

The feasibility of application of genetic algorithms for testing dependence was first proposed by Sudheer [3]. The solution as outlined there could provide only 'yes' or 'no' answer to the question of dependence between pairs of array references. Later Singhai [12] improved and extended it to determine direction vector for restructuring of programs. There are some other important aspects which are not addressed in any of the above work [3, 12]. These include,

- **Applicability of GAtest for unknown bounds.**

  Some loops involve symbolic constants or variables that may depend on input values as bounds. In both the cases, one or both the bounds of the corresponding loop index are unknown. The solution for unknown bounds is presented in Section 3.4.

- **Dependence direction for a trapezoidal search space.**

  Some restructuring techniques like privatization of variables need the direction vector. Singhai [12] addressed this problem for the rectangular bounds. It is not possible to extend his solution for the trapezoidal space in a trivial way. In Section 3.5.2 the problem of dependence direction for the trapezoidal space is considered and a solution is proposed.

- **Applicability of GAtest for S^2-search spaces.**

  A new concept called S^2-search spaces is introduced in Section 3.7. This problem is discussed in detail in Section 3.8.

3.1 GA approach to dependence problem

Recall the linear diophantine equation (2). Dependence problem, in view of the genetic algorithms, can be framed as a search problem that finds an integer point $(x_1, x_2, \ldots, x_n)$, called optimal point, in the search space defined by the loop bounds, that satisfies the linear diophantine equation.

Depending on the complexity of the constraints, the search space can be classified as *rectangular*, *trapezoidal*, *unknown bounds* or S^2-search space. These classes are not mutually exclusive. One class may fully or partially overlap another.

If a fitness value $\sum a_i x_i - a_0$ is associated with every integer point in the search space, then the problem is to obtain one of the fittest points, if one exists, in the search space.

The general approach of GAtest to solve the dependence problem, is as follows.

1. Apply Banerjee's test. If dependence is ruled out, there is no integer solution.

2. Use genetic algorithms to obtain a semi-integer solution **xp**, where **xp** $= (xp_1, xp_2, \ldots, xp_n)$.

3. If the semi integer solution obtained in step 2 has all integer elements, $xp_i \in \mathbf{N}$ $(1 \leq i \leq n)$, then **xp** is the solution vector. Otherwise divide the search space into two and apply GAtest to both resulting spaces.

Following sections explain these steps in detail for each of the special cases of the problem depending on the complexity of the constraints.

3.2 GAtest for rectangular space

Among all the classes of spaces simplest one is rectangular space with all known bounds. All the bounds are constants and do not involve any expressions or unknown variables. The bounds of this space can be represented as in equation (3).

Often Banerjee's test concludes dependence even though there is no integer solution. It was proved in [12] that Banerjee's test provides necessary and sufficient conditions for the existence of a semi-integer solution.

If Banerjee's test fails to rule out the existence of integer solution,

$$\exists \mathbf{X}\ \{\mathbf{X} = \{x_1, x_2, \ldots, x_n\}\ /\ \forall x_i \in \mathbf{R} \text{ and } \sum(a_i x_i) = a_0\}.$$

By applying genetic algorithm we can find a vector $\mathbf{X}$ such that

$$\forall i \neq k\ x_i \in \mathbf{N}, x_k \in \mathbf{R} \text{ and } \sum(a_i x_i) = a_0$$

Sometimes it is possible that x_k may also belong to $\mathbf{N}$. In that case integer solution is found. Otherwise the search space is divided into two subspaces whose lower and upper bounds are,

$$L_1 = \{l_1, l_2, \ldots, l_k, \ldots, l_n\} \qquad L_2 = \{l_1, l_2, \ldots, l_k = \lceil x_k \rceil, \ldots, l_n\}$$
$$U_1 = \{u_1, u_2, \ldots, u_k = \lfloor x_k \rfloor, \ldots, u_n\} \quad \text{and} \quad U_2 = \{u_1, u_2, \ldots, u_k, \ldots, u_n\}$$

Algorithm 3.1:

```
    GAtest(l, u, a) {
1       if (Banerjee_test(l, u, a) == TRUE) {
2           x = get_semi_integer_solution(l, u, a);
3           if (all xi are integers)
4               return(TRUE);
5           else {
6               let xi is the non integer element of x
7               tl = l;
8               tu = u;
9               tli = Ceil(xi);
10              tui = floor(xi);
11              return ( GAtest(l, tu, a) || GAtest(tl, u, a));
12          }
13      }
14      else return(FALSE);
15  }
```

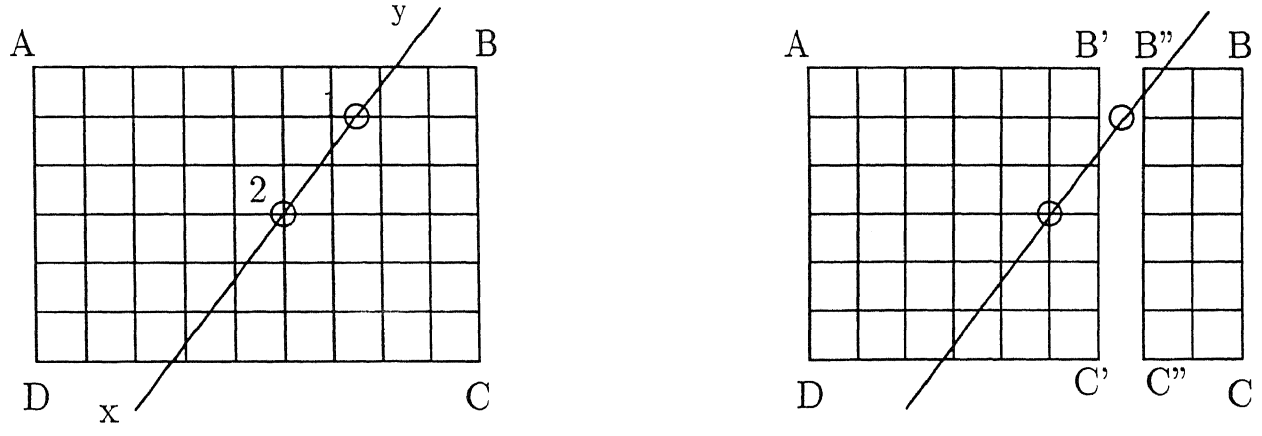


Figure 2: Division of two dimensional search space

In case if **X** is the only real solution within the search space then after applying to each of the subspaces Banerjee's test will rule out the possibility of a solution. This divide and conquer method is applied recursively until an integer solution is found or no subspace contains a solution.

Example 3.1: Consider the following linear diophantine equation,

$$3X_1 + 5X_2 = 10$$

and the constraints are

$$\begin{aligned} 1 \leq\ & X_1 \ \leq 10 \\ 1 \leq\ & X_2 \ \leq 10 \end{aligned}$$

Banerjee's test gives lower bound = 8 and upper bound = 35. So it predicts dependence. Application of GA resulted in semi-integer solution (1, 1.4). At this point GAtest divided the region {[1, 10], [1, 10]} into two subregions {[1, 10], [1, 1]} and {[1, 10], [2, 10]}. Banerjee's test again predicts dependence for the first subregion. Semi-integer solution (1.667, 1) is found using GA. It is divided further into {[1, 1], [1, 1]} and {[2, 10], [1, 1]}. Now Banerjee's test finds there is no solution to the given linear diophantine equation in both the subregions resulted at level 2. Application of Banerjee's test to the second subregion, at level 1, finds that there is no solution. So there is no integer solution to the given problem.

■

3.3 GAtest for trapezoidal search spaces

In trapezoidal search space, bounds of an index variable are linear expressions of the outer loops index variables. The general form of the constraints on the loop indices for this space can be represented as follows.

$$\left.\begin{aligned} l_{11} \leq\ & x_1 \ \leq u_{11} \\ l_{21} + l_{22}x_1 \leq\ & x_2 \ \leq u_{21} + u_{22}x_1 \\ & \vdots \\ l_{n1} + l_{n2}x_1 + \cdots + l_{nn}x_{n-1} \leq\ & x_n \ \leq u_{n1} + u_{n2}x_1 + \cdots + u_{nn}x_{n-1} \end{aligned}\right\} \quad (5)$$

GA approach to handle trapezoidal space is to first apply Banerjee's test. If the existence of an integer solution is not ruled out then a circumscribed rectangular space of the trapezoidal space is determined by adjusting the bounds to rectangular form. Following which a semi-integer solution is found similar to the rectangular spaces using GAtest for rectangular space. However, an integer solution for the circumscribed space may not satisfy the original constraints for trapezoidal bounds. So the feasibility of the solution must be checked against the original constraints. When the solution is a real solution the space must be partitioned into two. Unlike the rectangular spaces, a variable which contains other variables in its bound expression cannot be partitioned. So a check must be done sequentially from the beginning to locate a variable along which a division is possible. A variable should have more than one value to be divisible. If a variable can have only one value, then, it can be eliminated by substituting appropriate value in other variable's bound expressions as well as in linear diophantine equation. Though such an approach appears to be costly, in practice it is cheap. There will not be many variables and many of the subspaces will be eliminated by the Banerjee's test initially. So this is applicable.

3.4 GAtest for unknown bounds

In some cases the loop bounds can be symbolic constants. This may be true for both rectangular and trapezoidal spaces. Sometimes only one bound, either the lower bound or the upper bound, is unknown. At times both the bounds may be unknown. A variable whose both bounds are unknown is called *free variable*. In the previous chapter it was observed that many loops in subroutines are containing unknown bounds, because the bounds are containing some formal variable which is known only at the run time. In such cases dependence test must be able to decide whether the dependency exists with the known constraints. These unknown bounds can be assumed to be some finite value, like maximum value of the index of an array declaration for upper bounds and minimum value of the index for lower bounds. This is only an estimation of an unknown value, so this is not accurate enough.

GAtest assumes these unknown bounds to be, theoretically, $-\infty$ for lower bounds

and $+\infty$ for upper bounds. After assuming this Banerjee's test can be applied. If a free variable exists it is unnecessary to apply Banerjee's test, because it is known that the lower and upper bounds of the expression $\sum a_i x_i$ range from $-\infty$ to ∞. Even in other cases also it may happen that the bounds of the expression may range from $-\infty$ to ∞. It is known that if there are two free variables and their coefficients are relatively prime, then there always exists an integer solution.

If the dependence is not ruled out by the Banerjee's test, GA is applied to get semi-integer solution. In the application of GA, every time we assign a value to a variable we should check that the constraints of that variable, if these exist, are satisfied. While initializing the population, a free variable is assigned 0, which is the mid value of the limits assumed. If one of the bounds is known then it can be initialized with some random number that satisfies the known bound. If both bounds are known it can be initialized normally by a random value between lower and upper bounds. Algorithm to find the semi-integer solution is given below.

Algorithm 3.2:

```
get_semi_integer_solution(l, u, a)
l, u : Lower and upper bounds
a : linear diophantine equation.
{
   init_poulation(l, u);

   while (1) {
      apply_genetic_operator();
      x = select_a_member();
      j = random_number_in_range(1, n);
      xj= (a0 - sum(i!= j, ai xi))/aj;

      if (both bounds are unknown)
         return (semi_integer_solution = x)
      else if (satisfying_known_bounds(xi))
         return(x);
      else {
         if (lj_known && xj < lj)
            xj = lj;
         else   /* uj known and xj > uj */
            xj = uj;

         replace_worst_fit_member_by(x);
      }
   }
}
```

```
init_poulation(l, u)
l, u : Lower and upper bounds
{
   for (i=0; i<population_size; i++) {
      for (j=0; j<number_of_variables; j++) {
         if (both bounds known)
            pop[i]. var[j] = random_num_in_range(lj, uj);
         else {
            if (lower bound known)
               pop[i]. var[j] = random_number_above_or_equal(lj);
            else if (upper bound known)
               pop[i]. var[j] = random_number_below_or_equal(uj);
            else
               pop[i]. var[j] = 0 /* Best estimation */
         }
      }
   }
}
```

■

For all the following lemmas and Theorem, the dependence problem described by equations (2, 3) is assumed. We are proving that if there is only real solutions exist and there is no integer solution to the problem then GAtest detects it.

Lemma 3.1: *Let in the set of equations (3) some l_k is unknown. If there is no integer solution to the problem then x_k is bounded on the left side.*

Proof:

Let the limits of $\sum_{i \neq k} a_i x_i$ are C_{low} and C_{up}.

There are two different cases depending on the sign of a_k.

Case 1: $a_k > 0$

Lower limit of $\sum_{i=1}^{n} a_i x_i = B_{low} = -\infty$

Upper limit of $\sum_{i=1}^{n} a_i x_i = B_{up} = C_{up} + a_k u_k$

Linear diophantine equation has real solution so $a_0 \leq B_{up}$.

Eventually the region divided will result in a left subregion in which upper limit of x_k is

$$u'_k \leq \lceil \frac{a_0 - C_{up}}{a_k} \rceil - 1$$

now in the left subregion

$$B_{up} = C_{up} + a_k u'_k < a_0.$$

So the entire left subregion can be eliminated. Now the remaining right subregion has all known bounds.

Case 2: $a_k < 0$

Similar to the *Case 1*

$$B_{low} = C_{low} + a_k u_k$$

$$B_{up} = +\infty$$

When upper limit of x_k in the left subregion reaches a value

$$u'_k \leq \lceil \frac{a_0 - C_{low}}{a_k} \rceil - 1$$

then

$$B_{low} = C_{low} + a_k u'_k > a_0.$$

So the left subregion can be eliminated, resulting a problem in the remaining right subregion with known bounds.

In both the cases a left subregion is eliminated. And hence the x_k is bounded.

■

Lemma 3.2: *Let in the set of equations (3) some u_k is unknown. If there is no integer solution to the problem then x_k is bounded on the right side.*

Proof:

Let the limits of $\sum_{i \neq k} a_i x_i$ are C_{low} and C_{up}. Similar to the Lemma 3.1, this also has two cases depending on sign of a_k.

If $a_k > 0$. If the lower bound in the right subregion reaches $l'_k \geq \lfloor \frac{a_0 - C_{low}}{a_k} \rfloor + 1$, then the lower bound $B_{low} = C_{low} + l'_k > a_0$. So the right subregion can be eliminated. Similarly if $a_k < 0$, then if x_K reaches a lower limit $l'_k \geq \lfloor \frac{a_0 - C_{up}}{a_k} \rfloor + 1$, in the right

subregion then it can be eliminated,resulting in a problem in the left subregion of known bounds. Hence the lemma.

∎

Lemma 3.3: *Let both bounds of variable x_k in the set of equations (3) are unknown. If there is no integer solution to the problem GAtest detects it.*

Proof:

When the variable is divided at some random point it results in two subregions,

- a left subregion with unknown lower bound and known upper bound,
- and a right subregion with known lower bound and unknown upper bound.

These both cases are same as that are proved in Lemma 3.1 and Lemma 3.2. This results in the regions with known bounds. Hence from the correctness of GAtest [12] for known bounds, GAtest detects that there is no solution exists.

∎

Theorem 3.1 (Exactness of GAtest) *Let the linear diophantine equation be Equation (2) and bounds are (3) and some l_is and u_is are unknown. If the diophantine equation has no solution then GAtest proves that there is no solution.*

Proof:

Case 1: No integer solution exists for the diophantine equation in any region.

GCD test detects it. $d = \gcd(a_1, a_2, \ldots, a_n)$ does not divide a_0.

Case 2: If in the given region only real solution exists.

Some bounds of some variables unknown. From the division of the region when a variable with unknown bounds is divided it is known that it results in two subregions with same or less number of unknown bounds. Eventually this results in a case where only one variable has unknown bounds. For this case correctness of GAtest is proved in Lemma 3.3.

In the worst case if the same variable is divided always then one subregion may always remain with equal number of unknown variables as that of the original problem.

But in practice after reaching a bound value, that a computer can represent, it is no more possible to divide the same variable again. So this will result in division of the next possible variable. So eventually this results in a problem with less number of unknown values than the original problem.

Correctness of GAtest for known bounds was proved in [12]. Hence the theorem. ∎

In practice the variables of the linear diophantine equation take only finite values because the right hand side of the equation is a finite value. So if a solution exists then even though bounds are unknown, eventually the algorithm finds one of the finite solution points.

3.4.1 The worst case

Consider the following example.

$$\text{Linear diophantine equation } 2x + 4y + \cdots = 25$$

and bounds of the region are

$$\begin{aligned} -\infty \leq\ & x\ \leq +\infty \\ -\infty \leq\ & y\ \leq +\infty \\ & \vdots \end{aligned}$$

The $\cdots$ in the equation represent some more terms whose variables are of no interest. Assume that there is no integer solution. And a pseudo random number generator generated values that resulted in dividing always the variable x. Then the left subregion never terminates theoretically. But in practice, as the maximum and minimum of a number in any computer is fixed, in the worst case it will go up to that boundary. After reaching the boundary value it is no more possible to divide that variable so it continues with the next variable. Eventually the algorithm terminates. This is only a fictitious example. Often if more than two bounds are unknown and $\sum a_i x_i$ ranges from $-\infty$ to ∞ then integer solution exists.

Example 3.2: This example is same as that given in Section 3.2 except that the bounds changed as follows.

$$\begin{aligned} 1 \leq \quad X_1 \quad &\leq u_1 \\ l_2 \leq \quad X_2 \quad &\leq 10 \end{aligned}$$

Lower bound of X_1 and upper bound of X_2 are unknown. When Banerjee's test is applied after assuming MIN_INT for lower bound of the variable X_1 and MAX_INT for upper bound of the variable X_2 This gives both bounds of the expression $3X_1 + 5X_2$ are unknown. GA find a semi-integer solution (1.0, 1.4) satisfying the existing constraints. Now the region is divided into two subregions $\{[1, u_1], [l_2, 1]\}$ and $\{[1, u_1], [2, 10]\}$. At this point of time for the left subregion two bounds, equal to the number of unknown bounds in the original problem, are unknown and for the right subregion only one bound is unknown. The test is applied recursively. This is continued up to the depth of 6 levels where solution (5, -1) is found.

∎

If we compare the example in Section 3.2 with this example, even though the the later example seems to be searching in the region $\{[1, MIN\_INT], [MAX\_INT, 10]\}$, which is many thousands of times larger than the region of the former example, it took only about 2 times the time of the former example. In terms of number of invocations of GA, for the known bounds example it took 8 invocations where as for the unknown bounds example it took 22.

3.5 Testing for dependence for a given direction vector

Some restructuring methods need more information than is provided by simple GAtest to determine whether a pair of references causes the dependence. One of the important such information is a dependence direction vector. GAtest in its simplest form cannot generate information regarding dependence direction. However, This test may be called once for each of the possible combination of elements in a direction vector. The test have to be applied 3^n times to determine possible direction

vectors, if there are n variables. In this section we develop a method for finding dependence direction by modifying constraints as well as the diophantine equation. Both rectangular and trapezoidal search spaces are considered.

Given two statements are within a nest of loops of depth n. Dependence problem have to be solved for the direction vector $\mathbf{S} = (s_1, s_2, \ldots, s_n)$.

3.5.1 Rectangular search space

The set of original constraints for rectangular search space is,

$$l_i \leq I_i, J_i \leq u_i \text{ for } 1 \leq I \leq n.$$

- If s_i = '=':

  $I_i = J_i$. Therefore one of the I_i and J_i can be substituted by the other in the diophantine equation. This results in elimination of one variable. The constraints however will not change.

- If s_i ='<':

  In this case, search is to be confined to the space such that $I_i > J_i$. Thus the constraints for I_i and J_i should be changed to

$$\begin{aligned} l_i \leq \quad J_i \quad &\leq u_i - 1 \\ J_i + 1 \leq \quad I_i \quad &\leq u_i \end{aligned}$$

- If s_i ='>':

  In this case the target search should be such that $I_i < J_i$. That is, the constraints for I_i and J_i should be changed to

$$\begin{aligned} l_i \leq \quad I_i \quad &\leq u_i - 1 \\ I_i + 1 \leq \quad J_i \quad &\leq u_i \end{aligned}$$

3.5.2 Trapezoidal search space

Original constraints for both I and J are similar to the equation (5). Changing the constraints in this case is more complicated compared to the rectangular spaces. There are two different sets of constraints for I and J. Additional constraints, like $I_i = J_i$ or $I_i < J_i$ or $I_i > J_i$, have to be imposed in order to satisfy the given direction vector.

Consider the following example, one variable x_k having two constraints.

$$f(x_1, x_2, \ldots, x_{k-1}) \leq \quad x_k \quad \leq g(x_1, x_2, \ldots, x_{k-1}) \tag{6}$$

$$f'(y_k) \leq \quad x_k \quad \leq g'(y_k) \tag{7}$$

These constraints can be represented on a real line (we are interested for only the integer points of the line) that x_k takes.

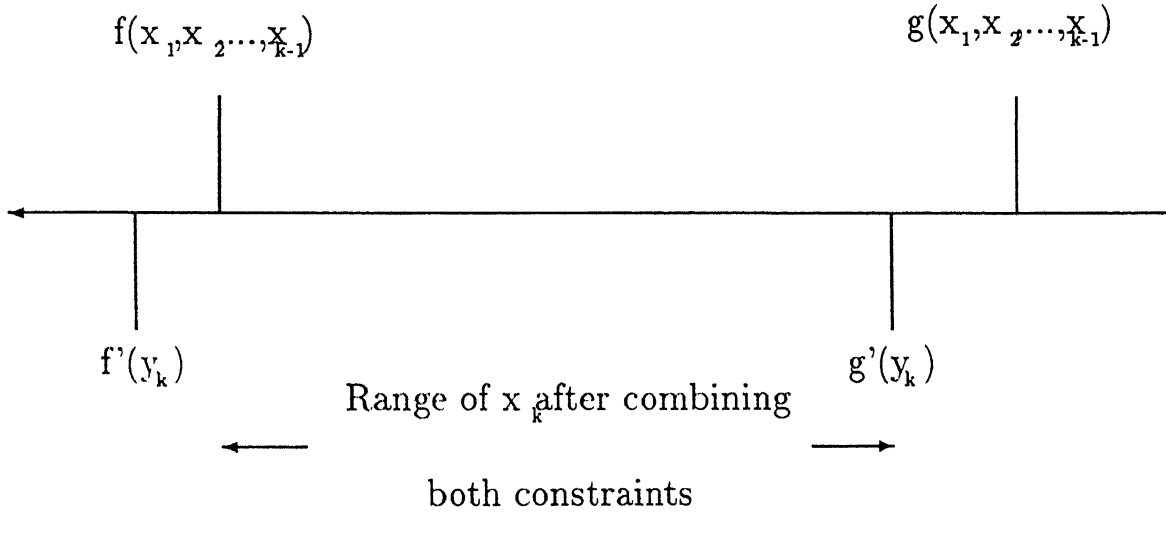


Figure 3: Variable having two sets of constraints

From the above figure actual range of values that x_k takes changes depending on the relative positions of points f, g, f', g'. Sometimes if there is no overlapping occurs between the intervals $[f, g]$ and $[f', g']$ set of values that x_k can take is *NULL*. i.e. space doesn't exist. Depending on the relative positions of the points, the lower and

the upper bounds of the x_k can be adjusted. In the above figure 3 new constraints on x_k are provided the following inequalities:

$$f(x_1, x_2, \ldots, x_{k-1}) \leq x_k \leq g'(y_k)$$

Mathematically this can be expressed as

$$\max(f(x_1, x_2, \ldots, x_{k-1}), f'(y_k)) \leq x_k \leq \min(g(x_1, x_2, \ldots, x_{k-1}), g'(y_k)). \quad (8)$$

Considering the problem of trapezoidal region,

- If s_i '=':

  $I_i = J_i$. One of the I_i and J_i can be substituted for the other in the diophantine equation and in the other constraints. the resulting problem has one variable less than that of the original problem.

- If s_i '<':

  In this case additional constraint $I_i > J_i$ has to be added. In the other form,

  $$J_i + 1 \leq I_i < u_{i0} + u_{i1}J_1 + u_{i2}J_2 + \cdots + u_{ii-1}J_{i-1}$$

  has to be added. The original and this new constraint can be combined together into one single constraint using equation (8).

  $$\begin{aligned} \max(J_i + 1, l_{i0} + l_{i1}I_1 + \cdots + l_{ii-1}I_{i-1}) \leq \ & I_i \\ \leq \min(u_{i0} + u_{i1}J_1 + u_{i2}J_2 + \cdots + u_{ii-1}J_{i-1}, & \\ u_{i0} + u_{i1}I_1 + u_{i2}I_2 + \cdots + u_{ii-1}I_{i-1}) & \end{aligned}$$

- If s_i '>':

  In this case additional constraint $I_i < J_i$ has to be added. Similar to the above case the new constraint is,

  $$I_i + 1 \leq J_i \leq u_{i0} + u_{i1}I_1 + \cdots + u_{ii-1}I_{i-1}.$$

Original and new constraints can be combined to give the following single constraint:

$$\begin{aligned} \max(I_i + 1, l_{i0} + l_{i1}J_1 + \cdots + l_{i,i-1}J_{i-1}) \leq\ & J_i \\ \leq\ \min(& u_{i0} + u_{i1}J_1 + u_{i2}J_2 + \cdots + u_{in-1}J_{i-1}, \\ & u_{i0} + u_{i1}J_1 + u_{i2}J_2 + \cdots + u_{in-1}J_{i-1}) \end{aligned}$$

3.6 Solution set enumeration

Sometimes it is necessary to enumerate all solutions. For example to find the distance vector. Most of the tests do not provide any information regarding the solution set or distance vector. Some restructuring techniques need the distance vector too, e.g. Conversion of sequential **do loop** to **do across loop**.

Finding distance vector in case of GAtest is done through enumerating the solution set. The search space is divided into three regions. Assuming that the current solution is $(x_1, x_2, \ldots, x_n)$ and the current search space is

$$\begin{aligned} x_1 \leq\ & I_1 \leq x_1 \\ x_2 \leq\ & I_2 \leq x_2 \\ & \vdots \\ l_k \leq\ & I_k \leq u_k \\ & \vdots \end{aligned}$$

the three regions of search space are:

Original and new constraints can be combined to give the following single constraint:

$$\begin{aligned}\max(I_i + 1, l_{i0} + l_{i1}J_1 + \cdots + l_{i,i-1}J_{i-1}) \leq\ & J_i \\ \leq \min(u_{i0} + u_{i1}I_1 + u_{i2}I_2 + \cdots + u_{ii-1}I_{i-1},\ & \\ u_{i0} + u_{i1}J_1 + u_{i2}J_2 + \cdots + u_{ii-1}J_{i-1}) & \end{aligned}$$

3.6 Solution set enumeration

Sometimes it is necessary to enumerate all solutions. For example to find the distance vector. Most of the tests do not provide any information regarding the solution set or distance vector. Some restructuring techniques need the distance vector too, e.g. Conversion of sequential **do loop** to **do across loop**.

Finding distance vector in case of GAtest is done through enumerating the solution set. The search space is divided into three regions. Assuming that the current solution is $(x_1, x_2, \ldots, x_n)$ and the current search space is

$$\begin{aligned} x_1 \leq\ & I_1 \leq x_1 \\ x_2 \leq\ & I_2 \leq x_2 \\ & \vdots \\ l_k \leq\ & I_k \leq u_k \\ & \vdots \end{aligned}$$

the three regions of search space are:

$$
\begin{array}{rcl}
x_1 \leq & I_1 & \leq x_1 \\
x_2 \leq & I_2 & \leq x_2 \\
 & \vdots & \\
l_k \leq & I_k & \leq x_k - 1 \\
l_{k+1} \leq & I_{k+1} & \leq u_{k+1} \\
 & \vdots &
\end{array}
\qquad
\begin{array}{rcl}
x_1 \leq & I_1 & \leq x_1 \\
x_2 \leq & I_2 & \leq x_2 \\
 & \vdots & \\
x_k + 1 \leq & I_k & \leq u_k \\
l_{k+1} \leq & I_{k+1} & \leq u_{k+1} \\
 & \vdots &
\end{array}
\qquad \text{and} \qquad
\begin{array}{rcl}
x_1 \leq & I_1 & \leq x_1 \\
x_2 \leq & I_2 & \leq x_2 \\
 & \vdots & \\
x_k \leq & I_k & \leq x_k \\
l_{k+1} \leq & I_{k+1} & \leq u_{k+1} \\
 & \vdots &
\end{array}
$$

The problem of solution enumeration through GAtest is also addressed in [12]. The original search space was divided into two subregions instead of three. Suppose the solution found is $(x_1, x_2, x_3, \ldots, x_n)$ and first divisible variable is I_k, then the two subregions according to [12] are,

$$
\begin{array}{rcl}
x_1 \leq & I_1 & \leq x_1 \\
x_2 \leq & I_2 & \leq x_2 \\
 & \vdots & \\
l_k \leq & I_k & \leq x_k - 1 \\
l_{k+1} \leq & I_{k+1} & \leq u_{k+1} \\
 & \vdots &
\end{array}
\qquad \text{and} \qquad
\begin{array}{rcl}
x_1 \leq & I_1 & \leq x_1 \\
x_2 \leq & I_2 & \leq x_2 \\
 & \vdots & \\
x_k + 1 \leq & I_k & \leq u_k \\
l_{k+1} \leq & I_{k+1} & \leq u_{k+1} \\
 & \vdots &
\end{array}
$$

GAtest will now be applied recursively on the two subregions. This does not generate the entire solution set as indicated by a simple example below.

Example 3.3: Let the diophantine equation be,

$$2x + 3y - 4z = 12$$

and the bounds of the variables be

$$
\begin{array}{rcl}
1 \leq & x & \leq 10 \\
1 \leq & y & \leq 10 \\
1 \leq & z & \leq 10
\end{array}
$$

Suppose at some point of time we got an integer solution (6, 4, 3). Now according to [12] division of region would result in two subregions

$$
\begin{array}{rcl}
1 \leq & x & \leq 5 \\
1 \leq & y & \leq 10 \\
1 \leq & z & \leq 10
\end{array}
\qquad \text{and} \qquad
\begin{array}{rcl}
7 \leq & x & \leq 10 \\
1 \leq & y & \leq 10 \\
1 \leq & z & \leq 10
\end{array}
$$

After applying GAtest recursively some more possible solutions may be obtained. However, if one observes carefully there is a solution (6, 8, 6) which does not belong to any of the two subregions. The reason is attributed to the fact that the specific value of the divisible variable which determined the partition is neglected. It is possible, for example, a solution, may exist where divisible variable has the same value in some other solutions for a different combination of values for the remaining variables.

■

But there is a problem with the technique of division of search space into three subregions. It may be possible that multiple instances of the same solutions is found repeatedly. In the third region the divisible variable has only one value. It can be substituted in the diophantine equation and can be eliminated. So total number of repetitions are limited to the number of variables.

In the following algorithm divisible variable means, it must be

- having more than one point,
- having no other variables in bound expressions.

Algorithm 3.3: For enumeration of solution set Line 4 of Algorithm 2.1. should be replaced by

```
    if (all xi are integers)
>   {
>       Solutionset = Solutionset + x;
>       j = select first divisible variable;
>
>       if (divisible variable found) {
>           tl = l;
>           tu = u;
>           tlj= xj+1;
>           tuj= xj-1;
>           GAtest(l, tu, a);
>           GAtest(tl, u, a);
>           Propagate first divisible variable value (in solution).
>           Form Resulting constraints ptl, ptu and
>                diophantine equation pa.
>           (p indicates values after divisible variable is
>              Propagated)
>           GAtest(ptl, ptu, pa);
>       }
>   }
    else
```

■

3.7 S^2-spaces

In most of the **do loops**, loop index gets incremented by some constant value for each iteration. If the loop increment is non unit the region described by the constraint equations will have some, invalid integer points. Loop control never goes through these invalid points. For example,

```
DO i=1, 20, 2
DO j=1, 20, 2
A(i, j) = . . .
ENDDO
ENDDO
```

```
DO i=1, 10
DO j=1, 10
    A(2*i, 2*j) = . . .
  ENDDO
  ENDDO
```

the loop constraints are,

$$1 \leq i \leq 20,$$
$$1 \leq j \leq 20.$$

The above loop execution does not goes through points (2, 2), (2, 4), ..., though the constraints show the existence of these integer points. Such loops with non-unit increments can be easily normalized to unit increment.

However, Some **while loops**, the general form of **for loops** in C programs usually have a recurrence relation of the index variable. This type of loops have many invalid points between the lower and upper bounds. Moreover these cannot be normalized to have any constant increment. Such type of search spaces, with a few valid points are called *'sparse search spaces'* or S^2-space.

3.7.1 Representation of S^2-spaces

In general, the recurrence relation of the index can be a polynomial expression. Lower and upper bounds need not be simple constants, they can be of trapezoidal form or triangular form. Let us assume that there are n nested for-loops with index variables $I_1, I_2, \ldots$. For simplicity assume that each loop's lower and upper bounds are linear expressions of index variables of outer loops(trapezoidal form) and recurrence relations are polynomial expressions of that index. Then the search space can represented as follows,

$$\left.\begin{array}{rrl} l_{00} \leq & I_1 & \leq u_{00} \text{ and } I_1 = r_1(I_1), \\ l_{10} + l_{11}I_1 \leq & I_2 & \leq u_{10} + u_{11}I_{11} \text{ and } I_2 = r_2(I_2), \\ & \vdots & \\ l_{n-10} + \cdots + l_{n-1n-1}I_{n-1} \leq & I_n & \leq u_{n-10} + \cdots + u_{n-1n-1}I_{n-1} \\ & & \text{and } I_n = r_n(I_n) \end{array}\right\} \quad (9)$$

where r_i is the recurrence relation of the index of i^{th} loop.

3.8 GAtest for S^2-spaces

None of the currently known test is applicable for S^2-spaces. Only GAtest by its nature of testing at points and dividing the region, can be adapted for S^2-spaces. Given the lower and upper bounds (l and u), recurrence relation (r) of a loop variable, define three functions on a number 'a' between lower and upper bounds,

1. $(a <=) = b$ such that $b \leq a$, $r(b) > a$ and b is in the sequence of numbers generated by the recurrence relation r.

2. $(a >) = b$ such that $b = r(a \leq)$ if $b < u$. Undefined otherwise.

3. $(a <) = b$ such that $r(b) \geq a$, $b < a$, if $b \geq l$ and b is in the sequence of numbers generated by the recurrence relation r. Undefined otherwise.

3.8.1 Distance between two numbers of a sequence

If a and b are any two numbers selected from the sequence of numbers generated by the recurrence relation r, number of valid points in search space in between a and b including b is the distance between a and b. We will denote the distance between a and b as *dist(a, b)*. Similar to the distance between two solutions in case of normal search spaces, this gives the distance in case of S^2-spaces.

3.8.2 Dependence problem in S^2-spaces

Let the search space is same as that described in Section 3.7.1. The diophantine equation to be solved is (2) The problem is to search for some integer solution to the equation (2), in the search space (9).

3.8.3 Finding integer solution

Constraints are trapezoidal so by using Banerjee's test we can find out the lower and upper bounds of the expression $a_1 I_1 + a_2 I_2 + \cdots$. If a_0 is in between the limits then Banerjee's test says there exists solution to the diophantine equation.

If Banerjee's test fails to rule out the dependence GAtest will be applied. The trapezoidal bounds are changed to bounds of a circumscribed rectangular space. Semi-integer solution is found using the following steps.

Finding semi-integer solution Given the rectangular bounds and recurrence relations, GAtest first initializes the population.

```
for (j=0; j<populationsize; j++)
{
    for (i=0; i<no_of_variables; i++)
    {
        rnum = random_number_in_the_range(lower[i], upper[i])
        member[j].var[i] = (rnum <=)
    }
}
```

It selects a member, and apply the genetic operations like mutation. Then the goodness of fitness for the resulting member is evaluated. The new member is included in the population by replacing the worst fit member.

```
member = select_member()
rnum = randdom_number_in_the_range(1, number_of_variables)
for all i except i = rnum find sum of ai * member.var[i]
```

```
member.var[i] = validated_var_value((a0 - sum)/a rnum)
if (have_enough_fitness(member)) include(member)
```

The validation of a variable is done through checking the value to be within the sequence of numbers generated by the corresponding recurrence relation. The value is first checked against the bounds and if it is within the bounds the (<=) operation is applied. Otherwise it is appropriately assigned to lower or upper bounds. If the value before validation is within the bounds then the solution is semi-integer solution. Otherwise the genetic operators are applied repeatedly until a semi-integer solution is found.

If the solution found is satisfying all the rectangular bounds then it must be checked against the original trapezoidal constraints. If the solution found is a semi-integer solution then it must be divided into two subregions. In order to divide the region, a variable satisfying the following constraints, is selected.

- Its lower and upper bounds expressions should not contain any other variable.
- $dist(lb_i, ub_i) > 1$.

Similar to trapezoidal regions, if lower and upper bounds of a variable are equal then that variable can be eliminated by propagating the value of that variable.

GAtest is applied recursively to the divided search spaces. This process is continued until an integer solution is found or all subspaces have no solutions. Detailed algorithm is given below.

Algorithm 3.4:

```
BOOL GATest(ltb, utb, diequ)
{
    if (BanerjeeTest(ltb, utb, diequ) == TRUE) {
        ConvertToRectangularBounds(ltb, utb, lrb, urb)
        SemiIntSol = FindSemiIntSol(lrb, urb, diequ)
        if (ValidIntSOl(SemiIntSol))
            return(TRUE)
        else {
            DivVar = SelectDivisibleVar()
            DevideSearchSpace(ltb, utb, ltb1, utb1, ltb2, utb2, DivVar,
                              Solution, diequ)
            Result = GATest(ltb1, utb1, diequ)
            if (Result == FALSE)
                Result = GATest(ltb2, utb2, diequ)

            return(Result)
        }
    }
    else return(FALSE)
}
```

```
Solution FindSemiIntSol(lrb, urb, diequ)
{
    InitPopulation()

    while (SemiIntegerSolution not found) {
        ApplyGeneticOperators()
        Member = SelectMember()
        VarNum = RandomValueBetween(1, NoOfVariables)
        Sum = For all i except i = VarNum
              CoefOf(i, diequ) * Member.Vars[i]

        TempVal = (ConstantTerm(diequ) - Sum) / CoefOf(VarNum, diequ)
        if (TempVal is in between lower and upper bounds of VarNum) {
            Member.Vars[VarNum] = TempVal
            Solution = Member.Vars
            return(Solution)
        }

        if (TempVal < lower bound)
            Member.Vars[VarNum] = lower bound
        else (TempVal > upper bound)
            Member.Vars[VarNum] = upper bound

        InsertIntoPopulation(Member)
    }
}
```

```
DevideSearchSpace(ltb, utb, ltb1, utb1, ltb2, utb2, DivVar, Solution, diequ)
{
    ltb1 = ltb
    utb1 = utb
    ConstantTerm(utb1) = (Solution[DivVar] <=)
    If (ltb1[DivVar] == utb1[DivVar])
        Propagate(DivVar, diequ, ltb1, utb1)

    ltb2 = ltb
    utb1 = utb
    ConstantTerm(ltb2) = (Solution[DivVar] >)
    If (ltb2[DivVar] == utb2[DivVar])
        Propagate(DivVar, diequ, ltb2, utb2)
}
```

■

Example 3.4: Consider the example having bounds

$$1 \leq I_1 \leq 100 \text{ and } I_1 = 2 * I_1 + 2$$
$$1 + I_1 \leq I_2 \leq 100 \text{ and } I_2 = I_2 + 1$$

and the diophantine equation is

$$I_1 + 2I_2 = 25$$

GAtest found a solution (1,12) within one invocation of the algorithm. Division of the region was not needed at all.

■

Similar to the normal trapezoidal search spaces in case of S^2-spaces also it is possible to enumerate the solution set by changing some portion of the algorithm. So it is possible to find out the dependence distance.

This idea can be extended to solve the non-linear equations too. In case of non linear equations if it is possible to find out the minimum and maximum values using some cost effective methods, if exists at all, then Banerjee's test can be substituted

with those methods. But this test is limited by the fact that in GA it must be possible for any variable to find out its value given values of remaining $(n-1)$ variables. In practice most of the array references are linear and the recurrence relations and lower and upper bound expressions are also linear.

3.9 Performance of GAtest

For the rectangular and trapezoidal bounds (normal search spaces) it was found in [12] cost of GAtest is

$$C(GA(l,u)) = \begin{cases} C(B) \text{ if Banerjee's test rules out the dependence,} \\ C(B) + C(GAsemi-integer(l,u)) + C(GA(l,tu)) + C(GA(tl,u)) \\ \text{otherwise,} \end{cases}$$

where (l,u) is bound of the search space, tl and tu are new bounds of the divided search space. In the worst case it is $O(N*c)$, where N is the size of search space and c is some constant mostly dependent on time required for finding semi-integer solution.

In case of S^2-search spaces also as the basic algorithm is same, constituents of the cost will be same, but the time required to find the semi-integer solution increases due the fact that every random value being assigned to a parameter of population must be checked or aligned to nearest valid point in the search space.

Following table compares the performances of GA and omega tests for the dependence problem of unknown bounds. Each test is applied 100 times in each case to get accurate time values. All times are shown in Millie seconds.

Table 5: Comparison of GAtest and Omega test

Program	*GAtest* Time for 100 executions(ms)	*Omega test* Time for 100 executions(ms)	Improvement of GAtest over Omega
sp29.t	102	142	28.1%
sp31.t	35	39	10.2%
sp43.t	25	30	25.0%
sp5.t	35	40	12.5%
sp50.t	25	31	19.3%
sp51.t	35	50	30.0%
sp52.t	68	89	12.4%
sp6.t	70	83	15.7%
sp65.t	25	35	28.6%
sp66.t	38	67	43.3%
sp7.t	42	47	10.6%
sp71.t	34	42	19.0%
sp72.t	35	40	12.5%
sp77.t	39	66	40.9%
sp79.t	41	59	30.5%
sp80.t	66	99	33.3%

Following table gives time requirements of GAtest for S^2-spaces.

Table 6: Timing of GAtest for S^2-spaces

Program	*Time (in ms)* for 100 executions
prob1	387
prob2	167
prob3	331
prob4	781

Chapter 4

WHILE Loop Parallelization

Most parallelizing compilers treat **WHILE** loops and **DO** loops with conditional exits as sequential constructs because their iteration space is unknown. In many fortran programs **if-then-go** type of loops exists in addition to normal **WHILE** constructs. In general the index variable may be a recurrence relation in **for** loops of C programs, in which case also iteration space is unknown. Some of these loops can be executed in parallel. Parallelization of **WHILE** loops is well studied in [5]. This work is done based on those techniques presented there in [5]. The implementation details are presented in Section 4.4.

4.1 Problems in parallelizing WHILE loops

In the most general form, a **WHILE** loop consists of the following three parts

1. one or more recurrences that can be detected at compile time,
2. a remainder whose dependence structure can be either analyzed statically or is unknown at compile time,
3. one or more termination conditions.

Assuming that there are no cross-iteration data dependences in the remainder, there are two potential problems in the parallelization of **WHILE** loops:

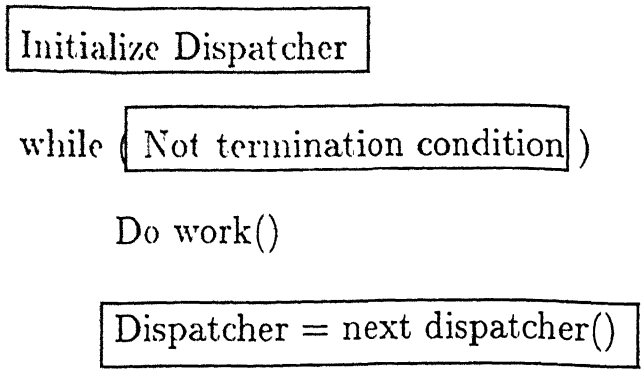


Figure 4: WHILE loop

- **Evaluating the recurrences:**

  If the recurrences cannot be evaluated in parallel, then the iterations must be started sequentially after evaluating the recurrences for each iteration, leading in the best case, to a pipelined execution.

- **Evaluating the terminating conditions:**

  If the termination condition cannot be evaluated independently by all iterations, the parallelized **WHILE** loop could continue to execute beyond the point where the original sequential loop would stop, i.e. it can overshot.

Data dependence analysis itself is a very complex problem in **WHILE** loop parallelization. Because iteration space is unknown, hence, it is difficult to perform dependence analysis at compile time. In [6] a run time technique called *Parallelizing DO ALL test* is proposed to analyze the data dependence problem with complex equations. This can be used for the dependence problem of **WHILE** loops whose iteration space is unknown. GAtest can handle some of the dependence problems of **WHILE** loops at compile time. This is presented in section 4.4.1.

Table 7: Taxonomy of WHILE loops and their dispatcher's potential for parallel execution. PP denotes parallelizable with parallel prefix computation.

Loop terminator	*Dispatcher*							
	Monotonic Induction		*NonMonotonic Induction*		*Associative Recurrence*		*General Recurrence*	
	Over shoot	*Parallel*	*Over shoot*	*Parallel*	*Over shoot*	*Parallel*	*Over shoot*	*Parallel*
RI	NO	YES	YES	YES	NO	YES-PP	NO	NO
RV	YES	YES	YES	YES	YES	YES-PP	YES	NO

4.2 Recurrence relations of a WHILE loop

A **WHILE** loop can have several dependent or independent recurrences. The dominating recurrence which precedes the rest of the computation in the dependence graph is called the dispatching recurrence or simply *dispatcher* [5] The dispatcher can be a simple induction or an associative recurrence relation. The induction dispatcher can be calculated independently using the closed form solution of the induction. The associative recurrence relation can be parallelized by parallel prefix calculation techniques. Sometimes dispatcher must be calculated sequentially, e.g. pointer used to traverse a linked list.

The terminator can be *remainder invariant* (RI), it is only dependent on the dispatcher and values that are computed outside the loop, or *remainder variant* (RV), i.e., dependent on some value computed inside the loop. If terminator is RV, then iterations larger than the last valid iteration could be executed in a parallel execution of the loop. The overshooting may also occur if the dispatcher is an induction or an associative recurrence and the terminator is RI. Depending upon the terminator type and dispatcher type **WHILE** loops can be classified as shown in Table 7.

Various techniques like *strip mining* [7] and *loop distribution* [5] and several optimizations for the distributed loops are proposed to parallelize the **WHILE** loops. Essentially techniques proposed in [5] work by distributing the original **WHILE**

loop into the two following loops:

1. A loop that evaluates the terms of the dispatcher and any termination condition that is strongly connected to the dispatcher.

2. A loop consists of the remainder loop and its associated termination condition.

For the optimizations on these distributed loops, for each type of dispatcher in Table(7), reader can refer to [5].

4.3 Undoing the iterations that overshoot the termination condition

The easiest way to undoing is to check pointing before loop execution starting and maintaining the record when a memory location is updated. When the loop is terminated, all the iterations below the iteration that caused termination will be copied back to original data. But this needs three times more memory than the actual memory necessary for data: (1) for check pointing, (2) keeping record of iteration number when it is modified and (3) original copy of data. This can be reduced if the data being updated by the **WHILE** loop is sparse by keeping only copies of those elements that were modified along with the time stamps. For this a hash table can be used. Thus total check pointing can be avoided.

A simple way to reduce the memory requirements is to strip mine [7] the loop, i.e., first execute s iterations then next s iterations and so on for a suitable value of s. In this case memory needed is bounded by the product of s and number of elements that were written in 1 iteration.

Time stamping can be avoided if the **WHILE** loop is run with check pointed data first and after determining the number of iterations, the loop can be executed simply as a **DO ALL** on original data.

4.4 Implementation

The program that is accepted by this module consists of declarations **if-then-else** statements , **WHILE** loops (not nested) and assignment statements. This module works in three phases.

- **Phase I:** Parsing

  In Phase I the parser parses the program is and constructs a syntax tree of the program. To generate the parser compiler construction tools *lex* and *yacc* are used.

- **Phase II:** Dependence analysis

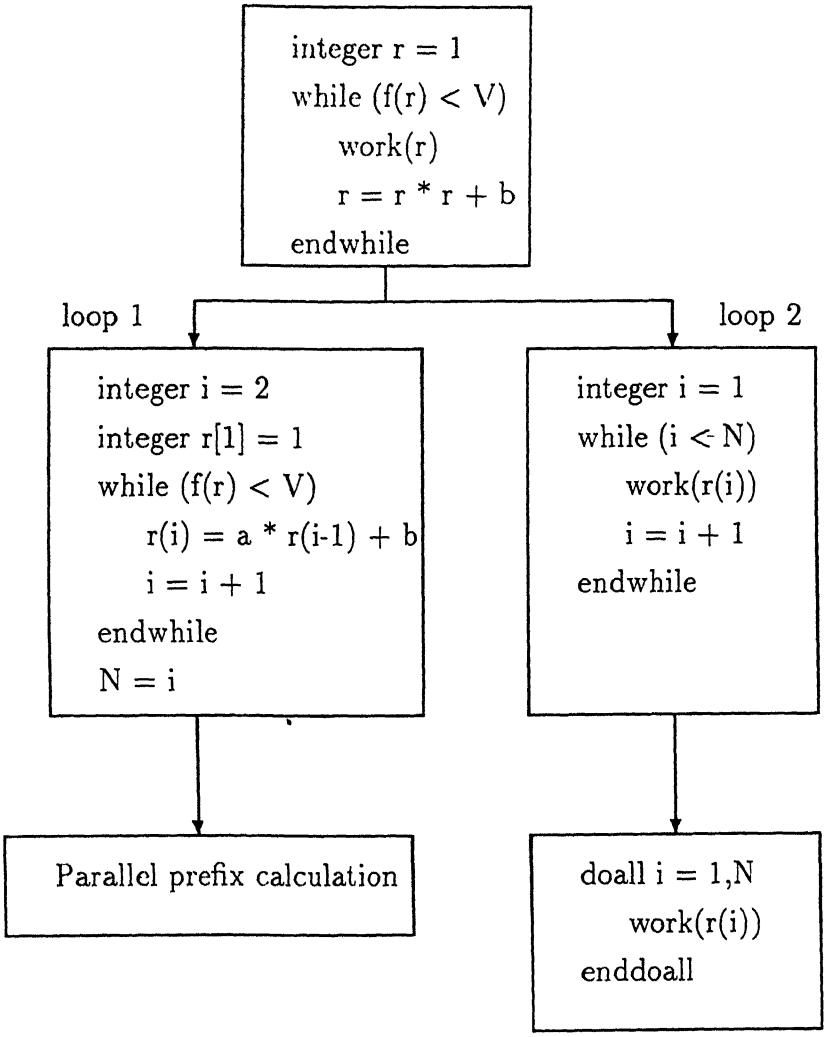
  In Phase II, a **WHILE** loop is identified and dispatcher is found. Initial value of dispatcher (reaching definition of dispatcher variable to **WHILE** loop) is found. It must be some known constant for the dependence problem to be analyzed at compile time. In some special cases like the dispatcher being incremented by 1, even though the dispatcher's initial value is not known, the dependence problem can be analyzed at compile time.

- **Phase III:** Code generation

  Intermediate code contains **DO ALL** statements in addition to that of source language. Parallelized distributed loops will be generated.

```
Analyze_dependences()
{
   for (all pairs of statements (s1,s2) inside the while) {
      common = (def(s1) ∩ def(s2)) ∪ (def(s1) ∩ use(s2)) ∪ (use(s1) ∩ def(s2));
      if (common != φ) {
         for (each possible dependence) {
            form dependence equations;
            call GAtest;
            if (dependence exists)
               return (dependence exists)
         }
      }
   }
   return (no dependence)
}
```

Example 4.1: Distribution of a **WHILE** loop.



■

4.4.1 Cross-iteration dependence testing of WHILE loops

Unlike the dependence problem of **do** loops the dependence problem of **WHILE** loops contains recurrence relations in addition to the constraints. The solution suggested in Section 3.8 can be used for the dependence problem. In order to apply the GAtest the subscript expressions must be linear and the lower bound of the

variables must be known.

Sometimes it is possible to execute the **WHILE** loop as a **WHILE ACROSS** (analogy to **do-across**) if there exist loop carried dependence and the dependence distance is of suitable size. If all the bounds of the variables are known, then the distance vector can be found using the GAtest.

In some cases of **WHILE** loops, it is not possible to analyze the dependence problem at compile time. This happens if subscripted-subscripts are used or subscript expressions are non-linear. In such cases execution of **WHILE** loop can be started in parallel. If the dependence is found at run time, it can be stopped and the loop will be executed sequentially.

If it is determined at compile time that there is no flow dependences and the only dependences are anti dependences or output dependences, then the loop can be executed in parallel by privatizing the variables that are accessed. A variable can be privatized if and only if all read accesses are preceded by write access to that variable. Even if it is not possible to analyze the dependence problem at compile time a **WHILE** loop can be parallelized using run time dependence checking called *privatizing DO ALL test* (PDtest) [6].

Except for the dispatcher of general case (dispatcher of linked list traversal program), GAtest is applicable in many cases. Even though the PDtest can be done in parallel, GAtest outweighs the former due to the cumulative time it consumes during many executions of the programs. Due to technical reasons author is unable to present the bench marking of GAtest for **WHILE** loops against known techniques.

4.5 Errors in parallel execution of WHILE loop

There are two types of errors that could occur during the speculative parallel execution of the **WHILE** loop: (i) exceptions (ii) the presence of cross iteration dependences in the loop. A simple way to deal with the exceptions is to stop the parallel execution, restore the previous values and execute the **WHILE** loop sequentially. In order to detect the cross iteration dependence in the parallel execution of **WHILE** loop, PDtest can be used. The PDtest is applied to each shared variable

that is accessed during the loop execution but whose accesses cannot be analyzed at compile time.

Chapter 5

Conclusions

Dependence problem is such an important part of a restructuring compiler which attracted the attention of many researchers. Many test were proposed. Most of the simple tests are conservative in nature where as accurate tests are complex to be used. From the study of some of the programs it was observed that in many cases a simple test like Banerjee's test is enough to solve the dependence problem. But the only problem comes from the analysis of subroutines which may contain formal variables as the loop limits. Singhai [12] suggested a suite of tests that can be used to handle all the cases. Based on the characteristics of the problem some of the test from the suite can be used.

GAtest is extended such that in almost all the cases it can be used. It requires less time compared to other exact tests. GAtest can provide information like dependence direction, dependence distance for a program restructurer. Only GAtest can provide information for optimal parallelizing of a program. i.e. by its capability of enumeration of dependent iterations pairs, it can be used to extract fine-grain parallelism.

Frequently observed unknown bounds problem in subroutines, can be handled by GAtest more efficiently than some other tests that can work with unknown bounds as shown in Table(3.9).

Many compilers treat the **WHILE** loops as sequential loops. In many cases these are parallelizable. Usually all such loops contains induction recurrence relation or

associative recurrence relation. Currently there is no compile time technique that can analyze the dependence problem involving recurrence relation. GAtest test is extended to handle this problem. Performance results proved that it can be included in a practical program restructurer. Rauchwerger and Padua [5] suggested a run time dependence analysis method. That works with any type of complex references. But in common cases it is unnecessary to postpone the dependence analysis problem up to run time which incurs overhead in each execution of the program. If the upper bound of the recurrence relation is also known then the GAtest can be used to exploit fine-grain parallelism in the case of while loops also leading to introduction of **WHILE ACROSS** loops in analogy to **DO ACROSS** loops. This is also helpful in determining the size of 'strip' in strip mining of the loop.

GAtest for unknown bounds can be extended to have symbolic constants in linear diophantine equation also. These constants can be treated as special variables whose value may be dependent on some loop limit of the region or simply a free variable.

GAtest can be extended to handle some non-linear equations too. There are two problems to extend this. These are:

- finding the bounds of the non-linear equation in the given search space,
- finding the value of a variable from the non-linear equation given the values for the remaining variables.

GAtest is limited by the second problem. After substitution of values of all variables except one variable in the non-linear equation, it will result in a polynomial. It is not practical to solve a polynomial of order more than 2. In practice such cases arise rarely. GAtest as it is now, is found to be more general and simple exact test.

Bibliography

[1] C.D.Polychronopolous. *Parallel programming and compilers.* Kluwer academic publishers, 1988.

[2] Gina Goff, Ken Kennedy, Chau-Wen Tseng. *Practical dependence testing.* Proceedings of the ACM SIGPLAN-91 conference on programming language design and implementation, Toranto, Ontario, Canada, June-1991.

[3] H.R.Sudheer. *GAtest: An exact dependence test for loop parallelization.* Master's thesis, I.I.T., Kanpur, 1993.

[4] Jose L.Ribeiro Filho, Philip C.Treleven, Cesare Alippi *Genetic algorithm programming environments.* IEEE Computer, June-1994.

[5] Lawrence Rauchwerger, David Padua. *Parallelizing WHILE loops for multi processor systems.* Center for super computing research and development, University of Illinois at Urbana Champaign, 1994.

[6] Lawrence Rauchwerger, David Padua. *The privatizing DOALL test: A run time technique for DOALL loop identification and array privatization.* Center for super computing research and development, University of Illinois at Urbana Champaign, 1994.

[7] Michael Wolfe. *Optimizing supercompilers for supercomputers.* The MIT press Cambridge, 1989.

[8] Michael Wolfe, Chau-Wen Tseng. *The power test for data dependence.* IEEE transactions on parallel and distributed systems, Vol. 3, Sept-1992.

[9] Michael Wolfe. *The Tiny loop restructuring research tool.* In Proceedings of 1991 International Conference on Parallel Processing, 1991.

[10] M.Srinivas, Lalit M.Patnaik. *Genetic algorithms a survey.* IEEE Computer, June-1994

[11] P.Suresh. *On classifying programs for data dependence tests in parallelizing compilers*, A technical report, Dept. of Comuter Science, I.I.T. Kanpur, 1995.

[12] S.Singhai. *Data dependence tests for loop parallelization.* Master's thesis, I.I.T., Kanpur, 1995.

[13] Utpal Banerjee. *Dependence analysis for supercomputing.* Kluwer academic publishers, London, 1988.

[14] William Pugh. *A practical algorithm for exact array dependence analysis.* Communications of ACM, Vol. 35, Aug-1992.

[15] Xiangyun Kong, David Klappholz, Kleanthis Psarris. *The I test:A new test for subscript data dependence.* International conference on parallel processing, 1990.

[16] Zhiyuan Li, Pen-Chung Yew, Chuan-Qi Zhu. *An efficient data dependence analysis for parallelizing compuilers.* IEEE Transactions on parallel and distributed systems, Vol. 1, Jan-1990.

SE-1996-M-SUD-1 xc.